LANGUAGE-INDEPENDENT PROGRAMMER'S INTERFACE

UNIVERSITY OF SOUTHERN CALIFORNIA

PREPARED FOR
ADVANCED RESEARCH PROJECTS AGENCY

MARCH 1974

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-73-15	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD 777957
4. TITLE *(and Subtitle)* Language-Independent Programmer's Interface		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert M. Balzer		8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, California 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2223/1
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, Virginia 22209		12. REPORT DATE March 1974 13. NUMBER OF PAGES 17
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* none 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution unlimited. Available from National Technical Information Service, Springfield, Virginia 22151.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

---

18. SUPPLEMENTARY NOTES

This paper will be given at the National Computer Conference and Exposition, Chicago, Illinois, May 6-10, 1974. It will also appear in the AFIPS Conference Proceedings, Vol. 43, AFIPS Press, Montvale, New Jersey, 1974.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

BBN-LISP and EL/1, domain-independent, human-engineered, interactive system, programming system, programming environment.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

PAGE V

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ARPA ORDER NO. 2223/1

ISI/RR-73-15
March 1974

Robert M. Balzer

Language-Independent Programmer's Interface

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308. ARPA ORDER NO. 2223/1, PROGRAM CODE NO. 3D30 AND 3P10

CONTENTS

ABSTRACT

A Programmer's Interface (PI) is a system which transforms an interpretive language into a programming system by providing a language-independent set of "environment" tools to be used in conjunction with the execution capabilities of the interpretive language. This "environment" consists of tools for creating, editing, debugging, filing and retrieving programs, for automatic spelling correction, for modifying and reissuing previous commands, and for undoing them to recover earlier states. A PI thus greatly expands the facilities available for program development without affecting the programming language or its capabilities. The importance of such a transformation cannot be overstated in terms of programmer productivity.

Any language with the following three properties can be interfaced to a PI at a fraction of the cost (several man-days versus several man-years) of creating a separate suitable programming system if: 1) there is a way to form a coroutine linkage between the language processor and the PI by interconnecting their I/O ports; 2) the language has an on-line evaluator and can field breaks or errors within a computation; and 3) either in such breaks or at the top level, the evaluator can evaluate arbitrary forms in that language.

A particular system (PI-1) has been constructed as an instance of the PI concept, using INTERLISP, and it provides INTERLISP's tools to interfaced languages. This PI has been successfully interfaced to ECL using the ARPA Network as the communications mechanism. The significance of this work lies in the observation that very little of the PI or the capabilities available in the INTERLISP programming environment are language-dependent, and in the experience gained in determining how a PI should be constructed and how languages should be interfaced to it, rather than in the interfacing between the PI and any particular language.

This work is of special relevance to large DOD-Military software production efforts. The research is directed toward higher productivity and higher quality software. This work is sponsored under ARPA Contract No. DAHC15 72 C 0308, ARPA Order No. 2223/1, Program Code No. 3D30 and 3P10.

INTRODUCTION

This paper addresses the general problem of creating a suitable on-line environment for programming. The amount of software, and the effort required to produce it, to support such an on-line environment is very large relative to that needed to produce a programming language, and is largely responsible for the scarcity of such programming environments. The size of this effort was largely responsible for the scrapping of a major language (QA4[1]) as a separate entity and its inclusion instead as a set of extensions in a LISP[2] environment. The few systems which do exist (e.g., LISP, APL[3], BASIC[4], and PL/I[5]) have greatly benefited their users and have strongly contributed to the widespread acceptance of the associated language.

At a bare minimum, a suitable programming environment consists of an on-line interpreter (or incremental compiler), an integrated interactive source-level debugging and editing system, and a supporting file structure. More extensive environments would include such facilities as automatic spelling correction, structural editors, tracing packages, test case generators, documentation facilities, etc.

Looking at several programming environment systems, one recognizes much uniformity. Most of the software supporting these systems is similar in both its organizational structure and functions. The systems differ in detail more from style differences between the system designers than from differences required by the programming languages.

The Programmer's Interface (PI) concept attempts to exploit this uniformity by creating a single programming environment capable of easily interfacing users with a wide variety of on-line programming languages. Users would then have the full facilities of this environment at their disposal. The PI is thus responsible for transforming these programming LANGUAGES into SYSTEMS. The cost of providing such an environment for a language would drop from the several man-years now required to the few man-days (estimated) to interface to a PI. Additionally, the existence of a common programming environment for many different languages would justify the inclusion of further capabilities.

This common programming environment provided by a PI should include facilities for: creating, modifying, storing, and retrieving programs; on-line debugging, including trace and break facilities as well as the facilities of the language for evaluation of expressions at breaks; modifying the interface between routines (via an ADVISE[6] capability); automatic spelling correction; remembering, modifying, and reissuing previous inputs; and undoing the effects of any of these PI facilities.

Such a PI has been constructed and interfaced to the programming language ECL[7]. The remainder of this paper explains the PI concept in terms of this implemented program. The deficiencies of this particular implementation are discussed in the conclusion.

SYSTEM ARCHITECTURE

The facilities provided by the implemented Programmer's Interface (PI-1) are based on the INTERLISP system (formerly BBN-LISP)[2]. In fact, they are the facilities of this system, as modified for language independence. The Programmer's Interface itself is implemented in INTERLISP and coexists with the facilities it invokes to provide the programming environment. INTERLISP was chosen as the basis both because it already had an extensive set of programming tools in an accessible form, and because their structure and operation could easily be altered to operate as required for a PI.

The system structure is shown in Figure 1. The ARPA Network[8] is used as the communications mechanism between PI-1 and the user's language processor. This choice has three advantages. First, it allows the interfacing of PI-1 to any language processor available on the ARPANET independent of what machine it runs on. Second, this interfacing can be done by PI-1 without the knowledge of the language processor. Thus no modifications to the language processor are required. Finally, the use of the Network greatly simplifies implementing the interconnection by allowing external character strings to be used for communication, rather than internal data structures with the attendant incompatibility problems.

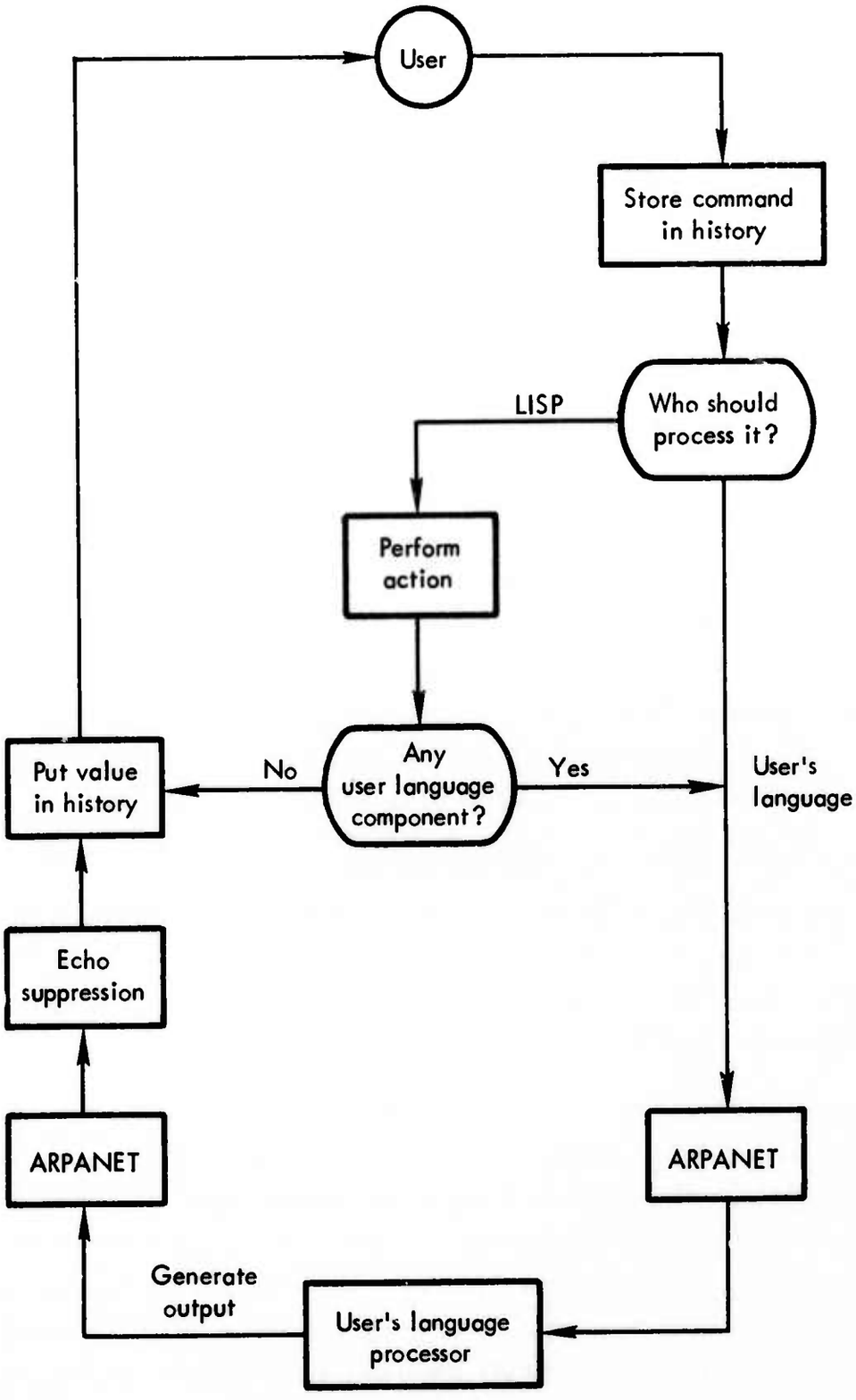

Figure 1. System architecture

Three properties are required of a language processor for its use with a PI:

1) There is a way to form a coroutine[9] linkage between the language processor and the PI by interconnecting their I/O ports. This type of linkage is discussed in detail in [10]. With PI-1, the ARPA Network provides this linkage. Thus, for PI-1, any language processor available on the ARPANET satisfies the first requirement.

2) It has an on-line evaluator (either an interpreter or fast compiler) and can field breaks or errors within a computation.

3) It can evaluate arbitrary forms in that language either in breaks or at the top level.

PI-1 begins processing user input by storing it in a history list used by the Programmer's Assistant[6], an INTERLISP subsystem, to retrieve, edit, group, reissue, or undo previous commands. PI-1 then examines the input to determine whether it should be processed by an INTERLISP facility or by the user's language processor. Basically, environment-type activities, such as loading files, editing programs, advising a function, etc., are performed within PI-1, while expressions in the user's language to be evaluated are passed to the language processor.

If the user's input is intended for his language processor, it is passed across the ARPA Network to that language processor. Any output generated by the processor is received across the Network again by PI-1. It suppresses the echo of the input and passes the output to the user, extracting from it the "value" and putting it into the history list for use by the Programmer's Assistant.

If the user's input is an environment-type command and should be performed within PI-1, the appropriate facility is invoked. In simple cases the operation completes, returns a value that is put in the history, and another input is processed. In more complex situations, some interaction is required during the operation with the user's language processor. This is accomplished by dynamically generating a series of inputs for the language processor that will have the desired effect or return the desired information. These are passed through the communications mechanisms to the processor;

its output is captured; and either the success of the modifications is verified or the desired information is extracted. Any number of such cycles may be required before the PI-1 facility completes its processing of the user's command. As an example, consider the loading of a file. As the function definitions are read in, they are stored as a property of the corresponding atoms to be used by the PI-1's editor for any modifications required later. The function definitions also are passed to the language processor so that it can use these for evaluation. Thus, one cycle is required for each function defined in the file.

PI-1 maintains a copy of all functions defined by the user and this is used by PI-1's editor when the user alters the definition. Whenever this definition changes (by redefinition or through exiting the editor), the resulting definition is passed to the language processor as a new definition of the function.

INTERFACING A LANGUAGE TO A PROGRAMMER'S INTERFACE

Most of PI-1 is language-independent, but certain portions must be modified to accept a new language. These fall into the categories of syntax modification, synchronization, program writing, and debugging.

The INTERLISP editor used by PI-1 is structural rather than string-oriented. To be effective, the text it is manipulating must have a structural basis. The syntax modification routines are responsible for introducing the structure into the user's language (only for use within PI-1). This structure is of two forms. First is the grouping of characters into lexical units. The user's language may have very different lexical grouping rules than LISP and the syntax modification package is responsible for the lexical analysis. Second, the lexical units thus produced are grouped into larger units by the use of parentheses. These units can be nested within one another to form the familiar LISP S-expression structure. The designer of the syntax modifier must decide where to introduce this structural grouping. In ALGOL-like languages, a natural place would be to group the lexical units of a statement together and groups of statements within blocks together. The structural groupings selected are introduced into all program text input by the user, and used by him to direct the editor in its

modifications of this text. When this text is passed to the language processor, those structural groupings artificially introduced for editing purposes are removed before transmission.

PI-1 and the language processor must be synchronized and kept in step with each other. Logically this is very simple and is accomplished by having PI-1 wait until the language processor has completed evaluating the previous input before giving it another. This situation is signaled by the language processor's attempt to read the next input. Unfortunately (due to a deficiency in the network protocol), this information is not available. Therefore the language processor's state of readiness must be determined by examination of its output stream. Fortunately, most on-line language processors explicitly indicate their readiness for more input by providing the user with a prompt character. The language processor's output must be scanned for this prompt and this is used as a synchronization mechanism between PI-1 and the language processor.

Several facilities within PI-1, such as break, trace, and advise, cause additional statements to be written into the user's program for evaluation at runtime. The interfacer of a new language must specify the form of these additions.

PI-1 contains many advanced debugging capabilities not found in most language processors. These aids are all based on information gathered during execution or at a break point within the program. To use these facilities, the designer of the language interface must supply routines that provide the basic information on which these debugging aids are built.

PI-1 took approximately three weeks to implement and debug, including the language interface to ECL. Although no other language interfaces have yet been built, it is estimated that an interface to another suitable language could be designed, implemented, and debugged in less than a week.

PI-1/ECL EXAMPLE

The following actual example indicates the use of PI-1 with the programming language ECL. The prompt character (as defined by ECL) is either ->, *, or a number

followed by :>. Commentary is enclosed in square brackets.

```
-> 3+4                          [Input of expression to be evaluated.]
7                               [Answer returned.]
-> TEST1<-EXPR(A:INT,B:INT,INT)BEGIN A+B; END;
                                [Define a function,TEST1, which takes
                                two integer arguments A and B and returns
                                their sum. Syntax is precisely as
                                defined for ECL.]
(TEST1)                         [TEST1 defined.]
-> TEST1(3,4)                   [Invoke TEST1 with arguments 3 and 4.]
7                               [Answer returned.]
-> EDITF(TST1)                  [Edit TEST1. Notice misspelling corrected by
                                system.]
= TEST1
EDIT
*PP                             [Prettyprint it. Notice how structure
                                has been added to its internal repre-
                                sentation.]
  (EXPR (A : INT , B : INT ; INT)
        (BEGIN (A + B)
               END))
*F BEGIN P                      [Find the item 'BEGIN' and print what
                                is found.]
(BEGIN (A + B) END)
*(2 (A GT B => A-B; A+B))
                                [Replace the second element, the list A+B,
                                by the remainder of the input. This is a
                                conditional form in ECL which evaluates
                                A-B if A is greater than B and A+B
                                otherwise.]
*PP                             [Prettyprint result. Again notice how
                                structure has been added.]
  (BEGIN (A GT B => (A - B))
         (A + B)
         END)
*(-4 (A=B -> B <- 2*A))
                                [Insert rest of input before fourth
                                element of current structure (the
                                END item). Addition says to set
                                B to 2*A if A=B.]
*PP                             [Prettyprint it.]
  (BEGIN (A GT B => (A - B)
         (A + B)
         (A = B -> (B <- 2 * A))
         END)
*UNDO                           [User notices his error (addition made at
                                wrong spot) and asks system to undo last
                                command.]
(-4 --) UNDONE.
*PP                             [Check to see that it's really gone.]
  (BEGIN (A GT B => (A - B))
         (A + B)
         END)
*USE -3 FOR -4                  [Substitute -3 for -4 in the in-
                                sertion command and reissue it.]
*PP                             [Make sure addition put in correct spot.]
  (BEGIN (A GT B => (A - B))
         (A = B -> (B <- 2 * A))
         (A + B)
         (END)
```

```
*OK                          [Exit editor.]
TEST1
-> TEST1(3,4)               [Test function. A is less than
                             B, just add them.]
7
-> TEST1(4,3)               [A greater than B, subtract
                             B from A.]
1
-> TEST1(4,4)               [A=B, double B and add in A.]
12
->ADVISE(TEST1 BEFORE (A<- 2*A))
                            [Modify TEST1 so that before it
                            is entered, but after its parameters
                            have been bound, the value of A
                            is doubled.]
TEST1
-> TEST1(3,4)               [Invoke modified function.]
2                           [Double 3 to get 6 and subtract 4.]
-> USE 6 3 10 FOR 4
                            [Successively substitute 6,3, and 10
                            for 4 in the last statement.]
18                          [TEST1(3,6)]
3                           [TEST1(3,3)]
16                          [TEST1(3,10)]
->ADVISE(TEST1 AFTER (VALUE\\ <- VALUE-1))
                            [Modify TEST1 so that after it is
                            finished, but before it returns,
                            the value to be returned is dec-
                            remented by 1.]
TEST1
-> REDO USE                 [Reissue the previous USE command
                            (which generated the 3 invocations of
                            TEST1)]
TYPE FAULT
- BROKEN
NIL
TYPE FAULT
- BROKEN
NIL
TYPE FAULT                  [3 type fault error occur.]
- BROKEN
NIL
3:>RETBRK(0)                [Go back to top level.]
NIL
-> TEST1(3,4)               [Try simple case.]
TYPE FAULT                  [Error still occurs.]
- BROKEN
NIL
1:> IN?                     [Where did error occur?]
IN ENTRY OR EXIT OF-
IN TEST1...
VALUE\\ <- VALUE - 1
                            [Error occurred in entry or exit of minus
                            routine which was invoked from TEST1 in
                            the statement VALUE\\<-VALUE-1. User
                            spots error (use of the undeclared
                            variable VALUE instead of VALUE\\).]
1:>EDITF(TEST1)             [Edit TEST1.]
EDIT
*F VALUE 0 P                [Find the use of VALUE. Go up one
                            structured level and print group.]
(VALUE\\ <- VALUE -1)
*R VALUE VALUE\\            [Replace VALUE by VALUE\\.]
```

```
*OK                          [Exit editor.]
TEST1
1:>TEST1(3,4)                [Try test case again.]
(1)                          [Double 3, subtract 4, then decrement
                             by 1.]
1:>^                         [Go up one level of error, in this
                             case to top level.]
NIL
-> REDO USE                  [Reinvoke previous USE command.]
(17)                         [TEST1(3,6)]
(2)                          [TEST1(3,3)]
(15)                         [TEST1(3,10)]
```

CONCLUSION

An extensive programming environment has been created for the ECL language through a program (PI-1) which allows the use of the already existing INTERLISP facilities. This greatly expands the user's facilities for creating, editing, and debugging his programs. His programming language has been transformed into a programming system. The availability of a comprehensive set of "environment" tools working in conjunction with the programmer's language is extremely important to his productivity.

The significance of this work, however, lies not in the particular interface provided between INTERLISP and ECL, nor in the extensive capabilities provided the user, but rather, in 1) the observation that very little of the interface itself, or of the capabilities provided, are language-dependent; 2) the recognition that the programming environment can be effectively split into an "environment" part and an execution and evaluation part; and 3) the experience gained from building such a system and interfacing a language to it.

PI-1, however, suffers from a number of deficiencies, the most important of which is the use of already existing tools in more general environments than they were designed for. This was most notable in the use of LISP's editor for nonstructured text (and the need therefore to introduce structure by parentheses) and the need to replace LISP's input routines to provide the proper lexical analysis for the interfaced language. Both of these problems could be avoided in a PI by having it use the syntax description of the language to guide the input, and editing and display of programs.

One of the strengths of the PI concept is the split between the "environment" part

and the evaluation part. This split, however, introduces the problem of communication and synchronization; each part must keep the other informed about changes it makes that affect the other. In PI-1, this communication and synchronization was partial and clumsy. The flow of information from the environment to the evaluation part was adequate, but the reverse flow was not. The need to communicate to another program suitable explanations of what the state of the evaluation was, what the cause of the error was, or even that an error occurred was simply not envisioned or planned for.

PI-1 has thus demonstrated that a moderately integrated PI can be built that has facilities far beyond what is typically available at a fraction of the cost. However, development of highly integrated PI will have to await a better understanding of the functional requirements of a language processor in such an environment.

Although the Programmer's Interface has only been interfaced to one language (ECL), and although it only contains a small fraction of the capabilities ultimately desired, it is having a major effect by acting as a prototype for a major software project [11,12] being undertaken to develop this understanding and provide a single, common, comprehensive programming environment interfaced to a wide variety of languages running on many different machines communicating through a network. New languages or machines could be interfaced to the system at a fraction of the cost of providing a separate programming environment. Widespread usage would justify the expenditure of more resources to augment and improve the capabilities provided. Such a PI could free users from having to develop their programs only with software available on their own machines and could provide a much more comprehensive and coordinated software development package than is currently available.

REFERENCES

1 Rulifson, J. F., J. A. Derksen, and R. J. Waldinger, QA4: A Procedural Calculus for Intuitive Reasoning, Stanford Research Institute, Artificial Intelligence Center, Technical Note 73, November 1972.

2 Teitelman, W. D., G. Bobrow, A. K. Hartley, and D. L. Murphy, BBN-LISP TENEX Reference Manual, Bolt Beranek and Newman, Inc., Cambridge, Mass., July 1971.

3 Falkoff, A. D., and K. E. Iverson, The APL Terminal System: Instructions for Operation, IBM Corporation, T. J. Watson Research Center, Yorktown Heights, New York, March 1967.

4 Kemeny, J. G., and T. E. Kurtz, BASIC Programming, John Wiley and Sons, Inc., New York, 1967.

5 IBM Corporation, C/S Time Sharing Option: PL/I Checkout Compiler, Form SC33-0033, November 1971.

6 Teitelman, W., "Automated Programming - The Programmer's Assistant," in the AFIPS Conference Proceedings, Vol. 41, Part II, AFIPS Press, Montvale, New Jersey, 1972, pp. 917-921.

7 Wegbreit, B., "The ECL Programming Systems," in the AFIPS Conference Proceedings, Vol. 39, AFIPS Press, Montvale, New Jersey, 1971, pp. 253-262.

8 Roberts, L. G., and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing," in the AFIPS Conference Proceedings, Vol. 36, AFIPS Press, Montvale, New Jersey, 1970, pp. 543-549.

9 Conway, M., "Design of a Separable Transition-Diagram Compiler," Communications of the ACM, Vol. 6, No. 7, July 1963, pp. 396-398.

10 Balzer, R. M., Ports - A Method for Dynamic Interprogram Communication and Job Control, The Rand Corporation, Santa Monica, Calif., R-605-ARPA, August 1971.

11 Balzer, R. M., T. E. Cheatham, S. D. Crocker, and S. Warshall, National Software Works Design, USC/Information Sciences Institute, RR-73-16, (in progress).

12 Balzer, R. M., T. E. Cheatham, S. D. Crocker, and S. Warshall, The National Software Works, USC/Information Sciences Institute, RR-73-18, (in progress).